Analyzing Parallel Program Executions Using Multiple Views

Thomas J. LeBlanc
John M. Mellor-Crummey
Robert J. Fowler

CRPC-TR90037
January, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Analyzing Parallel Program Executions Using Multiple Views*

Thomas J. LeBlanc†
Computer Science Department, University of Rochester, Rochester, NY 14627

John M. Mellor-Crummey‡
Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251

Robert J. Fowler
Computer Science Department, University of Rochester, Rochester, NY 14627

Rice COMP TR90-110
January 1990

---

*To appear in *Journal of Parallel and Distributed Computing*, June 1990.

†Supported in part by the National Science Foundation under research grant CCR-8704492, and an Office of Naval Research Young Investigator award, contract no. N00014-87-K-0548.

‡Supported in part by the National Science Foundation under research grants CCR-8704492 and CCR-8809615.

Abstract

To understand a parallel program's execution we must be able to analyze lots of information describing complex relationships among many processes. Various techniques have been used, from program replay to program animation, but each has limited applicability and the lack of a common foundation precludes an integrated solution. Our approach to parallel program analysis is based on a multiplicity of views of an execution. We use a synchronization trace captured during execution to construct a graph representation of the program's behavior. A user manipulates this representation to create and fine-tune visualizations using an integrated, programmable toolkit. Additional execution details can be recovered as needed using program replay to reconstruct an execution from an existing synchronization trace. We present a framework for describing views of a parallel program's execution, and an analysis methodology that relates a sequence of views to the program development cycle. We then describe our toolkit implementation and explain how users construct visualizations using the toolkit. Finally, we present an extended example to illustrate both our methodology and the power of our programmable toolkit.

1 Introduction

Parallel programming is hard. Not only must programmers deal with new and unfamiliar abstractions for managing parallelism, they must also make do with tools designed to support the development of sequential programs. Debugging is complicated by the need to examine and understand abstractions for expressing parallelism, communication, and synchronization, and yet traditional cyclic techniques for debugging may not help, since many parallel programs exhibit non-repeatable behavior. Improved performance is usually the reason for creating a parallel program, but program profilers are of little use in performance tuning, since the impact of an individual process on overall performance may be hard to measure, the effects of process decomposition difficult to discern, and the outcome of specific optimizations impossible to estimate. The ineffectiveness of tools designed for sequential programming and the lack of tools specifically designed for parallel programming are the primary impediments to the development of parallel software.

In both debugging and performance tuning, the information to be analyzed is voluminous, including the states of each process and interactions among processes, and often describes complex relationships among components of the execution. Different *views* of the execution can be constructed by emphasizing or ignoring selected information. For each view, there are many different ways to present the information contained in the view, including varieties of both graphical and textual displays. A *visualization* makes some details of the view manifest, while obscuring others. A view defines *what* information is presented; a visualization describes *how* the information is displayed.

In an ideal program analysis environment, each aspect of anomalous program behavior would be revealed by some view wherein the cause is manifest. No one view is sufficient; any particular view contains either too much or too little information to understand some aspects of a program's behavior. Unless the programmer knows exactly what to look for, interesting phenomena can be lost in the sheer volume of information. What we require is a unified approach to parallel program analysis that supports the creation and integration of multiple views of an execution.

Unfortunately, most recent efforts in program analysis are problem-oriented; tools are developed to address one specific class of problems, such as deadlock, erroneous synchronization, or hot-spot contention. Problem-oriented approaches do not generalize to other errors or situations; a tool for visualizing phases in an application provides little or no insight into application performance. This lack of generality limits the applicability of each individual tool, and gives us little confidence that all situations encountered during program development will be equally supported by some tool.

Program analysis would be greatly simplified if those tools that do exist could be used together. However, most tools are architecture or language specific and lack a common foundation, making it difficult to migrate among tools. Even the addition of a common foundation is not sufficient, since an unstructured collection of individual tools provides no guidance about which tool should be used in a particular situation. Tools should be structured so that a programmer can select views of a program execution in some reasonable sequence, where each view takes into consideration previous views and the programmer's current needs.

This paper describes our approach to parallel program debugging and performance analysis, which uses multiple views for analysis of an execution. We monitor a program execution to collect synchronization traces, which form the basis for analysis. From these traces, we build a representation for an execution that, along with deterministic execution replay, enables construction of multiple views of the program's behavior. A programmable toolkit is used to create and fine-tune visualizations based on these views.

In the following section we outline our view framework and suggest an analysis methodology based on a series of views for an execution. We describe our toolkit implementation and explain how users construct visualizations using the toolkit. Finally, we present a case study that illustrates both our methodology and the power of our programmable toolkit.

2 View Framework

A problem-oriented approach to parallel program analysis is unlikely to address all situations uniformly or lead to an integrated tool base for parallel software development. Therefore, rather than emphasize a particular class of problems, we define our framework for analysis in terms of the information to be analyzed. This approach has three advantages. First, by defining the execution information available for analysis, we create a common information base for tool development. Second, we can use the dimensions of the information space to create associated views; varying individual dimensions in the information space suggests how new views can be created from known views. Finally, the set of available views can be used to define a presentation order that forms the basis of a methodology for analysis.

Any type of analysis is limited by the information that is made available. Our particular approach to monitoring is to capture a fine-grain characterization of an execution based on a trace of the synchronization events that occur. Such an approach applies equally well to loosely-coupled processes that communicate using message passing, or tightly-coupled processes that communicate via shared memory. In our implementation for shared-memory programs [12], each process records a history of its operations on locks protecting shared data structures. The union of the individual process histories specifies a partial order of operations on each shared data structure. This partial order, together with the source code and input, characterizes an execution of the parallel program and is referred to as an *execution history*. Histories can be augmented with user-defined events for application-level debugging and time stamps for performance analysis. Program replay based on histories can be used to reproduce executions that can be examined at an arbitrary level of detail, providing access to the values of variables and the state of individual processes.

During the execution of a parallel program, multiple asynchronous processes interact and make internal state transitions as time progresses. Parallel program analysis, therefore, requires that we navigate through a three-dimensional space of views, in which each point is characterized by its treatment of *process interactions*, *process states*, and *time*. We must be able to examine these dimensions both collectively and in isolation. At a given moment in time, we can view the state of an execution as the union of the states of each process. For a given process, we can trace its execution using local state transitions over time. Given a globally consistent state, such as arrival at a barrier, we can determine when (in time) each

process arrives at that state. Each of these viewpoints requires that we focus on a different dimension; taken together, these viewpoints paint a complete picture of an execution.

These three dimensions form the basis of our framework for describing views of an execution. The framework is complete, since it addresses all aspects of a program's execution. The framework is also general, since it only defines the available information, not how the information is selected or presented. In fact, current techniques for parallel program analysis and visualization can be categorized within this framework. In the following sections, we expand on the meaning of each dimension in the framework.

2.1 Process Interactions

Processes can interact in many different ways. The most obvious and most studied interactions are explicit communication and synchronization, including message passing, rendezvous, remote procedure call, semaphores, and monitors. Other, more subtle forms of interaction, involve shared resources, such as data, files, or processors. Both forms of interaction are often the root cause of errors, but no tool designed for sequential programs can be of much help in analyzing these interactions.

When analyzing a program's execution, the focus of interest often begins with the entire program, and then moves to smaller groups of processes, or even a single process. The collective behavior of the processes defines the results of the program, but the behavior of individual processes may be the cause of anomalies. When considering many processes simultaneously, the interactions among processes are of primary concern. Of necessity, as more processes come under consideration, a more abstract representation of each process must be used. Thus, the number of processes in the current focus of interest limits the information that can be made available and defines the extent to which global behavior can be analyzed.

In the most abstract view, called a *program view*, the effects of process interactions are seen, but the interactions themselves are hidden. This view corresponds to the behavior of the entire program; the results of the algorithm are visible, but not its multi-process implementation.

In an *anonymous process view* the program is seen as a collection of unstructured processes. This view allows relatively coarse parallelism analyses, but provides no information about process relationships or interactions. Summary statistics about parallelism are an example of an anonymous process view; the indentity of processes contributing to the summary is not important.

Additional insights into the behavior of the program require a more detailed presentation of processes and their interactions. An *interprocess communication view* shows communication relationships among processes. These relationships can be static or dynamic, depending on the hardware, programming environment, and application. A static view is used for describing message channels between processes; a dynamic view describes the actual transmission of messages or accesses to shared data.

A *data parallelism view* illustrates a relationship between processes and data. Although some parallel programs are relatively unstructured, logical relationships are often formed using data parallelism, wherein the structure of the data is used to form process relationships.

For example, a large matrix might be divided among many processes; a data parallelism view describes which process controls each portion of the matrix.

A *processor view* shows a relationship between processes and physical processors. Performance can be greatly affected by the binding of processes to processors and the physical connections between processors. For example, an appropriate binding of processes to processors might allow the logical connections among processes to be implemented directly with physical connections. Analyzing communication performance may require knowledge of the logical connections among processes, the physical connections between processors, and binding of processes to processors. By combining a processor view and an interprocess communication view, the requisite information is selected.

Each of these views can be presented in a variety of ways. To illustrate process interactions, both explicit and implicit, it is necessary to embed the interactions in a 2-dimensional display medium. For example, communicating processes organized in a grid can be presented in a grid [8]; hypercubes of processes can be embedded in a plane [3]. To illustrate the physical relationships, we can superimpose the logical relationships onto a presentation of the underlying architecture.

2.2 Process State

An individual process is a sequential program. Tools designed for sequential program analysis can be used to examine the state of processes. State-based uniprocess debuggers, such as dbx, and sequential program animation systems, such as Balsa [2] and PECAN [14], provide the type of information required to examine the state of individual processes.

As with other sequential programs, we can view the state of a process at several levels of abstraction. An *external view* contains an abstract description of the process in terms of its external behavior. The precise information available in this view is application-specific; the output of a program is an example of an application's external state.

Although external state is useful in determining whether or not a process is functioning as intended, it is not sufficient to understand exactly how a process works. An *infrastructure view* contains a description of how the process computes values, rather than solely what values are computed. This view includes the values of local variables, the current instruction being executed, the procedure call chain, and other state information normally accessible to a single process debugger.

In a shared system, scheduling of a process or job mix, satisfying memory needs, and resolving shared resource contention is usually transparent to user programs. However, to maximize performance, an entire multiprocessor is often dedicated to the execution of a single parallel program at a time, in which case the specific choices made by the scheduler, memory manager, or resource allocator might be under user control. These decisions can significantly impact the execution of a program and are frequently the subject of analysis. A *system view* considers the state of the underlying system as part of the application. Processor utilization and the state of memory allocation are examples of information visible in the system view.

2.3 Time

Time is a crucial dimension that must be considered in the analysis of parallel programs. It is often as important to know when an event occurred as it is to understand the exact state change that took place as a result of the event. Two distinct concepts of time are often used. Physical time defines the duration of an entire execution and individual operations within an execution. Logical time (as defined by the *happened before* relationship [10]) defines the observable order of events with a causal relationship. In a *logical time view* we can determine only whether an event occurs before or after another; using a *physical time view*, we can determine how much time passes between two events. Physical time, therefore, is more detailed than logical time. Given a physical time scale for all events in an execution, we can construct a logical time scale, but not vice versa.

Physical time measurements require a common clock or synchronized clocks, whereas logical time can be easily propagated from event to event. As a result, we typically use physical time to measure the duration of an operation within a process, and relate events in different processes using logical time. Logical time is crucial to understanding the correctness of a parallel program, while some form of physical time base is needed during performance analysis.

A third concept of time, which we call phase time, is also useful. Phase time orders a sequence of events with respect to collections of related events, or phases, that occur during execution. For example, if a program exchanges data among processes in well-defined rounds of communication, the set of communication events in a round form a phase. Each communication event occurs during some phase, and all phases are totally ordered with respect to phase time. Phase time is an example of a *synthetic view* [2], since there might not be a mapping from phase time to program variables. A precise definition of program phase is application-dependent, therefore, phase time requires a specification of the relationships that constitute a phase. Given such a specification, a *phase time view* can be constructed from logical time; the definition of a phase can be used to resolve the *happened before* relationship for events that do not have a causal relationship in logical time. This restriction on logical time can be used to structure an asynchronous execution into synchronous phases.

The way in which time is presented, either physical, logical, or phase time, can significantly affect the ease of program analysis. In particular, the time dimension can be presented either spatially or temporally, corresponding to "current" and "history" respectively in Brown's *persistence* dimension [2]. Process time lines [7, 10, 17] use space to present time, translating the temporal dimension of a program's execution into a spatial dimension in an execution graph. In this representation, events that occur together in time are grouped together in space, and are separated in space from events that occur either earlier or later. In addition, if a physical time base is available, long intervals of time can be represented using more space than is used for short intervals.

Animation systems [3, 8, 9, 16], on the other hand, present time temporally by dividing an execution into time frames. A program is represented by a static structure, usually a regular communication structure [3, 8], onto which frames of interesting events are superimposed over time. Each frame contains events considered to occur simultaneously in time;

later events are contained in a frame that appears later in the sequence.

To date animation systems have used frames constructed from phase time [8, 16] and logical time [9]. Frames based on phase time are particularly useful for analyzing communication patterns in the application. Progress in an execution animation is measured in terms of program phases familiar to the programmer; events that occur within a single phase are presented simultaneously in the display. Frames based on logical time are also suitable for analyzing communication operations, although no attempt is made to group events based on application knowledge; progress in the animation is seen from the system level rather than the application level. In both cases the granularity of temporal relationships shown in a single frame is limited because two events at the same location cannot be shown simultaneously. As a result, either the frame definition cannot exactly mirror our intuitive notion of an application's phases, or some events must be deleted from each frame. In addition, these animation systems are not useful for performance analysis, which requires physical time.

Presenting time spatially also has problems in that, depending on how the processes are placed in space, the logical communication patterns may or may not be obvious. For example, the sequence of program phases illustrated by animation may be obscured in a spatial presentation by irrelevant events that appear to occur simultaneously with the phase behavior. On the other hand, since parallel program analysis in general, and performance tuning in particular, requires detailed examination of temporal relationships, a representation that makes those relationships explicit using a spatial dimension can facilitate analysis.

2.4 Classifying Related Work

Our framework for describing views of parallel programs can be used to classify current tools based on the views they support. Both BALSA [2] and PECAN [14] are animation systems for sequential programs. BALSA presents a program view primarily as a teaching aid; PECAN includes several different infrastructure views for debugging. Neither tool was designed for parallel programs, so interprocess communication, data parallelism, and processor views are not supported.

Voyeur [16] is a parallel program animation system, similar in flavor to BALSA. Voyeur presents an application-specific program view in phase time. Depending on the application, a data parallelism view can be inferred from the display. No support for infrastructure or system views is provided, and the lack of a physical time view precludes performance analysis with this tool.

Belvedere [8] is designed to illustrate patterns of communication in parallel programs. Interprocess communication is presented in phase time. The underlying architecture is assumed to be reflected in the process connections, so both processor views and data parallelism views can be inferred from the interprocess communication view. However, more abstract program views and less abstract infrastructure and system views are not supported, and the restriction to phase time precludes both detailed analysis of event orderings and performance tuning.

BugNet [18], Radar [11], and Jade [9] are all monitoring systems for distributed programs. Each presents a view of interprocess communication in logical time, although BugNet

has mechanisms for dealing with physical time as well. The external behavior of processes is the primary focus of concern; single process debuggers are used to examine infrastructure. Program and anonymous process views are not provided directly; the emphasis on distributed systems obviates the need for a data parallelism view.

PIE [13, 15] is designed for performance debugging of parallel programs. It supports several different views in physical time, including program, anonymous process, and processor views. These views are integrated with infrastructure and system views to trace the cause of performance problems. The focus in PIE is on performance tuning, so interprocess communication views, phase time views, and logical time views, which are mainly useful for debugging, are not supported.

3 A View-Based Methodology

Our framework for program analysis defines the problem space of interest and different views of that problem space. The framework structures the information to be made available, but does not define any particular sequencing of views. Our proposed methodology fills that role, defining the order in which analysis problems should be attacked and the views to be used in each case.

Two general principles have guided our development of a methodology: (1) analysis follows the program development cycle, progressing from error detection and diagnosis to performance analysis and tuning, and (2) analysis proceeds top-down, from abstract views to more specific views. By organizing views from our framework into a sequence based on the program development cycle, the programmer examines a series of views, beginning with the program's abstract functional behavior, progressing through low-level views of internal states of processes, and ending with detailed timing characteristics for performance evaluation. Transitions in the sequence correspond to advances in the program development cycle, from initial debugging to performance tuning.

There are four significant phases in our methodology corresponding to stages in the program development cycle. The first two phases are generally associated with debugging; the last two phases deal with performance issues. In the first phase, we are interested in understanding whether the program works and how it works. The process of choosing appropriate test inputs is beyond the scope of our work, but in this phase we must interpret the output produced when the program is executed using selected test inputs. Our goal is to determine the correctness of the algorithm in terms of its abstract behavior.

In the second phase, we are interested in discovering the causes of the behavior seen in the previous phase. We must consider the internal behavior of the program over successive executions to either discover problems not visible from the abstract behavior or to more completely diagnose the cause of erroneous behavior. Our goal is to produce what we see as a correct program, thereby ending the current debugging cycle.

In the next phase, the general performance characteristics of the application are examined, including the degree of parallelism and the overall running time. Once again, our goal is to simply understand the performance of the program, regardless of the cause. Finally, in the last phase, we tune the performance of the program, using successive executions to

develop and validate a detailed model of the program's performance. This model is used to suggest changes to the program to enhance performance and to extrapolate the program's performance to larger inputs or a larger number of processors.

Each of these phases emphasizes different information in the execution space and can exploit several different views of the same information. Our methodology is an attempt to structure the views derived from our framework among these analysis phases. Broadly speaking, we iterate over the dimensions of the view space, from the more abstract views to the specific views, varying process interactions first, followed by process states and time.

3.1 Error Symptoms and Causes

Error diagnosis and correction is fairly well understood for sequential programs. Repeated experiments with test inputs are used to generate and recognize error symptoms; controlled execution with state examination is used for diagnosis. Our methodology for parallel programs is no different. We use multiple views to help recognize symptoms and program replay both to control execution and for state examination.

Debugging begins with an understanding of the execution as represented by the output results. At this stage inputs are chosen either to exercise certain execution paths through the code, or to produce structured results that may be easily checked. The simplest view of a program's external behavior is its raw textual output. While this may prove adequate when results are expressed in terms of a few key values, a typical parallel program operates on a large data set and a user can be easily overwhelmed by the volume of output. The role of scientific visualization [5] is to create a pictorial representation for the results of a computation. One specific technique for visualizing parallel programs is to capture user-defined events for display in an application-specific format [16]. Depending on how this application-specific program view is constructed, both symptoms and causes may be apparent in the related visualization.

To determine how results are computed, we need to examine interprocess communication. Phase time is used if the program has meaningful phases that exhibit patterns of communication. A logical time view can be used if the program lacks meaningful phases, or if additional details not provided by the phase time view are required. Both error symptoms and causes related to interprocess communication can be seen with these views.

Once the cause of an error has been isolated within a process, we can use a single process debugger to provide an infrastructure view of that process. Program replay with single-stepping allows us to examine the process's state at any level of detail.

3.2 Performance Analysis and Tuning

To improve the performance of a program, we must first understand its performance characteristics. The first step is to determine the overall running time of the program. Although indicative of the program's performance, the running time is primarily useful as a basis for evaluating the utility of future optimizations.

To determine where a program spends its time in terms of the program's phases (assuming they exist), we use a combination of phase time and physical time. These views

show how the running time is distributed across the phases, enabling the programmer to determine whether the running time is distributed throughout the phases as expected and which phases dominate the running time.

The next step is to determine the ratio of communication to computation, either within a phase or within the entire program. When the ratio is small, computation dominates and sequential code must be improved to significantly affect performance. When the ratio is large, too much of the execution time is devoted to information exchange rather than computation. Several different views are used to analyze this ratio.

First, an anonymous process view is used to determine the global communication and computation ratio. Assuming that the global ratio is relatively large for the application, an interprocess communication view is used to determine the ratio for each process. If the ratios for all processes are roughly in balance, the granularity of computation is too small, and the process decomposition strategy must be modified (or the cost of communication must be reduced).

If the program as a whole is dominated by the communication costs of a few processes, we need to examine those processes in greater detail. An interprocess communication view in physical time will help show any trends in the communication costs, which can then be correlated with the algorithm. If no such trends are apparent, then communication can be correlated with problem partition, scheduling, and load balancing using system and processor views.

When the percentage of execution time devoted to communication and synchronization is acceptably small, we must tune the sequential portions of the algorithm to significantly improve the overall running time. The decision as to where best to apply optimizations is more complicated than for sequential programs. First, the primary cause of poor performance may be distributed across many processes, so profiles of program code may not be of much help. Second, communication and synchronization may hide the effects of certain optimizations; some optimizations may not improve performance at all. Critical path analysis [19] is needed to determine what sequential code should be improved in order to improve overall program performance. Therefore, the next view of the program should contain the critical path embedded in the dynamic interprocess communication graph. Since the critical path will include both process interactions and sequential code, it must be possible to embed the path in either an external view or infrastructure view. Once the critical path has been determined, a standard profiler can be used to determine where the sequential code should be improved. As new performance enhancements are introduced into the program, the emphasis in analysis shifts back and forth between interprocess communication and sequential code, until a satisfactory balance is achieved.

During performance analysis and tuning, the programmer can build a performance model of the program's execution. The various views help provide the parameters to the model; constants can even be measured using a physical time view. Such a model can help predict the execution time of the program for larger inputs or a greater number of processors.

4 Toolkit Implementation

Our methodology for analyzing parallel program executions involves presenting the programmer with a series of views ranging from global views of program state to detailed views of characteristics that affect a program's performance. To facilitate this style of analysis, we are developing an integrated toolkit that supports management and presentation of information about a program's dynamic behavior [7]. Components of the toolkit are built around a representation of a parallel program execution that is synthesized by integrating trace information from each of the processes in the execution. Here we describe our approach to execution monitoring, how we integrate trace information to build a representation of an execution, and how a user can manipulate this representation to create views using our toolkit's programmable interface.

4.1 Execution Monitoring

To support analyses of a parallel program's dynamic behavior, it is necessary to gather information about an execution of the program. During execution, we collect a trace of all of the synchronization events that occur [12]. On shared-memory multiprocessors, synchronization events typically correspond to the use of locks to coordinate accesses to shared data structures. A trace indicating how processes interact over time through access to shared data structures can provide important information about the program's algorithmic structure, the effectiveness of the problem partitioning strategy used, and the level of parallelism exploited, among other properties.

The monitoring overhead incurred by a program is a function of how often synchronization primitives are invoked, which is determined by both the granularity and frequency of sharing. For our synchronization tracing technique to be efficient, the granularity of synchronized access to shared data structures must be larger than an individual memory location. Otherwise, if these structures are accessed frequently, synchronization overhead, including the recording costs, would dominate during execution. Our approach is most suited to programs with coarse-grain sharing in which a shared data structure protected by a lock might be a row (or set of rows) in a matrix, a message buffer, or a task queue. In these cases the relative infrequency of lock acquisitions, compared to data accesses both local and shared, often results in monitoring overhead of less than 3%.

To capture a synchronization trace, programs must use calls to instrumented synchronization primitives, provided by a library package, to perform all synchronization operations. Primitives in the library record an event record in a trace file each time a primitive is used. When an instrumented program is executed, each process records a private trace file of the synchronization events in which it participates. Each event record in a process's trace history contains a type code for the operation, a unique id specifying the synchronization variable accessed, and a sequence number for the synchronization variable. These fields in an event record enable the toolkit to relate events in different process traces. Each event record also includes one or more time stamps that provide temporal information about the event. Instantaneous events, such as relinquishing a lock, are annotated with a single time stamp that indicate when the event occurred. Events with extended duration are annotated

with two or more time stamps: the time that the process first requests access to the object, the time that access is granted, and the time that access is relinquished.

In addition to the event records that are automatically recorded for an execution trace, a programmer may annotate a program to generate tag records corresponding to other interesting events. The library primitive supplied for this purpose records a user-defined 32-bit quantity with a timestamp. Typically these records are used to label different activities or phases of activity by a process. Although we have not done so, instrumentation could also be added to the standard I/O libraries to generate trace records to indicate delays incurred by processes as a result of I/O operations. Also, the operating system scheduler could be instrumented to generate additional trace records (probably in a separate trace file per processor) that detail process time-slicing activity.

4.2 Constructing an Execution History Graph

Traces collected during execution serve as input to our toolkit. One of the most important functions of the toolkit is to merge individual process trace histories to form an execution history for the program. An execution history is represented as a directed acyclic graph in which each node in the graph represents an event. Events within a process are linked by directed arcs denoting the order of the events in the sequential execution of the process. Directed arcs between events in different processes denote temporal precedence relations between operations on a single synchronization variable. Some arcs represent a pure synchronization constraint (or *anti-dependence*), such as the requirement that the contents of a buffer be removed before the buffer can be refilled. Other arcs represent the transfer of information (a *true dependence*), such as the requirement that a buffer be filled before its contents can be read.

To build an execution history graph, the toolkit reads the sequence of events in the trace history of each process from its trace file, marks each event with the identifier of the process to which it belongs, and links it into a chain representing the sequence of events for that process. If the event record represents a synchronization event, a pointer to the event is placed in a list corresponding to the synchronization variable accessed by the event. After the event trace for each process has been read into the toolkit, the events associated with each synchronization variable are independently sorted by their sequence number. Following specifications given by a configuration file, the toolkit adds dependence edges between events on the same synchronization variable that have identical or adjacent sequence numbers.

To understand the addition of dependence edges to an execution history graph, consider an implementation of a concurrent-reader-exclusive-writer lock supporting four types of operations: **readstart**, **readend**, **writestart**, and **writeend**. The lock implementation maintains a monotonically increasing sequence number that indicates the current version of the data structure guarded by the lock. Each **writestart** operation increments the sequence number before recording it in the operation's event record. All other operations record the current sequence number of the object in their event records. To add dependence edges to the execution history graph that relate write operations to subsequent read operations, a specification must be added to the configuration file indicating that a directed edge should

be added to the execution history graph from a **writeend** event to all **readstart** events on the same synchronization variable with the same sequence number. Similar specifications can be added to indicate dependences between **readstart** and **readend** events, as well as those between **writestart** and **writeend** events.

4.3 Creating Views of a Program Execution

The base layer of the toolkit is responsible for creating and maintaining execution history graphs. The toolkit directly supports creation of a number of views based on this representation. In addition, a programmable interface based on Kyoto Common Lisp [20] provides toolkit users with the capability to define and manipulate visualizations of an execution history graph, as well as the capability to extract and analyze information represented in the graph. Views extracted using the programmable interface can then be used as input to other utilities for visualization. Here we describe how the components of the toolkit can be used to create views of an execution.

A central component of the toolkit is Moviola [6], an interactive execution browser for execution history graphs. Moviola displays an execution history graph as a space-time diagram. The two dimensions in a Moviola diagram correspond to processes and time. Events are arranged in vertical columns according to the process to which they belong. Each process column is arranged with the earliest event at the top and the lastest event at the bottom. The layout of events in a column can be selected to use either logical time or physical time. In a physical time view, events are displayed as boxes with height proportional to their duration, and spacing proportional to differences in event timestamps. A shaded box representing an event indicates time the process spent waiting during the event. In a logical time view, events are displayed as hash marks with spacing based on a topological layering of all events in the execution. Arcs between events in a space-time diagram indicate precedence relationships. Each arc between a pair of events is implicitly directed from the event nearer the top of the display to the event nearer the bottom. Arcs between adjacent events in the same vertical column represent the temporal ordering of events in the sequential execution of an individual process, while each arc connecting events in different processes indicates a temporal relationship between a pair of operations on the same synchronization variable. Figures 2–4 and figure 8 (in the following section) show space-time visualizations of execution history graphs created by Moviola.

Moviola provides the user with considerable interactive control of the execution history graph visualizations. Zooming, panning, and scrolling in both dimensions can be used to navigate through a graph to scan for phenomena of interest. Moviola also provides options that allow a user to select subsets of processes, event types, or synchronization objects for display. If more extensive control is necessary, display attributes of nodes in a graph can be set using the Lisp-based programmable interface. For example, a Lisp utility to compute the critical path can traverse the execution graph and mark each of the nodes along the critical path to be highlighted by Moviola. Similarly, a utility could highlight display of all descendants of a particular node to show the transitive causality of events corresponding to Lamport's *happened-before* relation.

The space-time diagrams provided by Moviola are visualizations of only one type of view

of a program execution. For convenience of interpretation, a user might prefer to extract simpler views from this representation and present them using other visualizations. For example, a user might want to determine the average amount of parallelism during an execution. Although this could be inferred (with considerable effort) from a Moviola diagram by carefully measuring the time that each process spent waiting during the execution (as indicated by shaded bars in a Moviola physical-time diagram), it is much simpler to write a utility program to traverse an execution history and automatically compute this sort of statistic. The toolkit provides an abstract interface for all aspects of an execution history graph. Using the abstract interface, utility programs written for the toolkit can access the attributes of individual nodes in the graph, as well as traverse the graph using ancestor and descendant information. The Kyoto Common Lisp environment enables users to create customized analysis and visualization tools in interpreted Lisp, compiled Lisp, or any other language that can be compiled into relocatable code. Interpreted Lisp is convenient for interactive analysis and testing; compiled Lisp runs much faster and is preferable for production tools. Other languages are used when a program written in another language is imported into the toolkit. Using the programmable interface, we are able to construct new views and their corresponding visualizations as they are needed. Section 5 presents visualizations of a variety of views extracted from our execution representation using utility programs written in Lisp.

One class of views that is not directly constructible from our execution history graph representation are views that contain information about a process's local state during execution. Fortunately, such state information can be reconstructed upon demand and imported into the toolkit. The synchronization traces we collect during our monitoring phase characterize the behavior of a program execution and can be used to replay the execution on demand. A synchronization trace can be used by our library of synchronization primitives in a subsequent execution to guarantee that an equivalent ordering of accesses to shared data structures occurs. Assuming that the input to the program from external sources (*e.g.*, files, or keyboard input) is identical, insuring that processes interleave their accesses to shared data structures in an equivalent manner is sufficient to guarantee that each process in the execution will pass through the same sequence of states. (A more detailed discussion of using synchronization traces for execution replay appears in [12].)

Using program replay, we can augment an execution history with whatever additional data is required to create a particular view. For example, our toolkit could use program replay to collect more detailed information about process states, such as the value of shared variables at a particular point in the program. Access to such information can be invaluable for debugging. By integrating symbolic debugging capabilities with existing components of our toolkit, one could replay an execution up to a selected point, probe the state of the execution, and create a corresponding visualization. A completely integrated system would allow users to specify breakpoints in an execution history by selecting events in a Moviola display. These abstract breakpoints would be converted into low-level commands to halt the execution at the proper point.

Time and process interaction aspects of a view are selected by performing programmed traversals of the execution history. These traversals build the view by collecting and aggregating the data from the execution history along various cuts through the DAG. Visu-

alization entails projecting the data onto an appropriate set of dimensions in the display medium.

Many families of views are generated by examining data from each process trace along a sequence of cuts across the execution history, each cut corresponding to an instant in time. By varying the degree to which the data is aggregated across these cuts we obtain program, anonymous process, and interprocess communication views. By varying the model of time used to define the cuts we obtain phase time, logical time, and physical time views. Each of these views can be constructed using simple Lisp routines to traverse the execution history. For example, the degree of parallelism at each instant of time can be obtained by summing the number of active processes across cuts corresponding to instants in physical time.

Once the information in a view has been collected, it can be passed on to visualization tools for projection onto the display medium. Deciding exactly how to display a view entails choosing from among many alternatives. Because there are so many valid choices for the visualization of each view, the toolkit does not dictate any one particular style of visualization. Users may massage the data using application-specific tools or the data can be passed to a standard analysis package, such as S [1], for interpretation and display. The toolkit is designed to be both flexible and extensible, so that, with modest effort, users may create new views and corresponding visualizations.

5 Applying the Methodology: A Case Study

In this section we illustrate the use of our methodology by applying it to the analysis of a parallel program for the upper triangulation step of Gaussian elimination. The program represents a system of linear equations as a matrix of coefficients. The jth pivot step entails subtracting a multiple of the jth row (referred to as the *pivot row*) from each of the higher-numbered rows in the matrix, creating a simpler linear system with one less unknown. Since these subtractions are independent of one another, they can be done in parallel. The program partitions an $n \times n$ matrix evenly among p processors by assigning rows to the processors in a modular fashion. Each processor i manages those rows k, $0 \leq k < n$, for which $k \ mod \ p = i$.

Message passing is used for communication between processes. When a pivot row is complete, it is broadcast to all other processes by placing it in an output buffer, where it is available to other processes. Each process that needs the row copies it from the output buffer into a local buffer.

The program is structured as a master process and a set of worker processes, one per available processor. At startup, the master creates the worker processes. Each worker performs its initialization code and synchronizes with the master. When all workers have completed their initialization code, the master releases them to begin the computation. When the computation is complete, workers synchronize with the master and are subsequently destroyed.

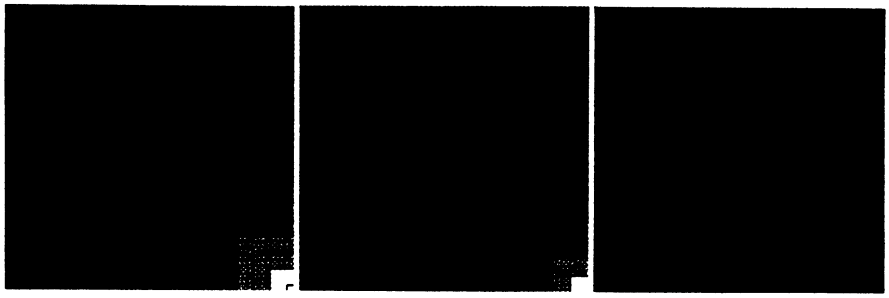

Figure 1: Contour Plots Showing Progress of Upper Triangulation.

5.1 Debugging

The first phase in the analysis is to determine whether the program is functionally correct. This typically involves examining the external behavior of the program for selected inputs. We can inspect the progress of upper triangulation by producing a sequence of displays of the coefficients of the matrix. If the row and column indices of the matrix are plotted in the $(x - y)$ plane, the coefficients can be interpreted as heights (in the z dimension). As triangulation progresses, column elimination causes the values of matrix elements below the diagonal to become 0, causing the corresponding portion of the surface to collapse in the z dimension. One visualization of this effect is a perspective plot showing a projection of the surface. Another visualization is a contour plot in which each element is assigned a color or shading based on its value.

Figure 1 shows three program views of the upper triangulation using contour plots. The contours are spaced at even intervals to account for the range of the data. The leftmost contour map represents the initial linear system (a 200×200 matrix) which is defined as:

$$mat(i,j) = \begin{cases} 2(j+1) & j < i \\ 2(i+1) & otherwise \end{cases}$$

This input data was chosen to produce a known output, including a visualization with a simple, recognizable structure that remains regular throughout the triangulation. The shaded bands distinguish elements that are grouped into the different contours. The shading is such that the lower the value, the darker the shading. The middle contour map is a snapshot at the mid-point of the computation. The elements below the diagonal in the first 100 columns have been reduced to 0. The elements in the rows that have been completed take on the value 2 and, as expected, a solid band of shading appears above the diagonal. The rightmost contour map shows the completed triangulation of the matrix. All of the elements below the diagonal are 0 and the other elements are all 2.

If the program view had shown that the computation was not progressing properly, we could have moved to a more detailed view of the execution to examine the communication patterns. Several types of visualization can help. The programmable interface can be used to recognize application-defined phases in the execution history to produce either snapshots or an animation of the communication activity. Space-time diagrams of process synchronization, such as those produced by Moviola, can also can be used to display the

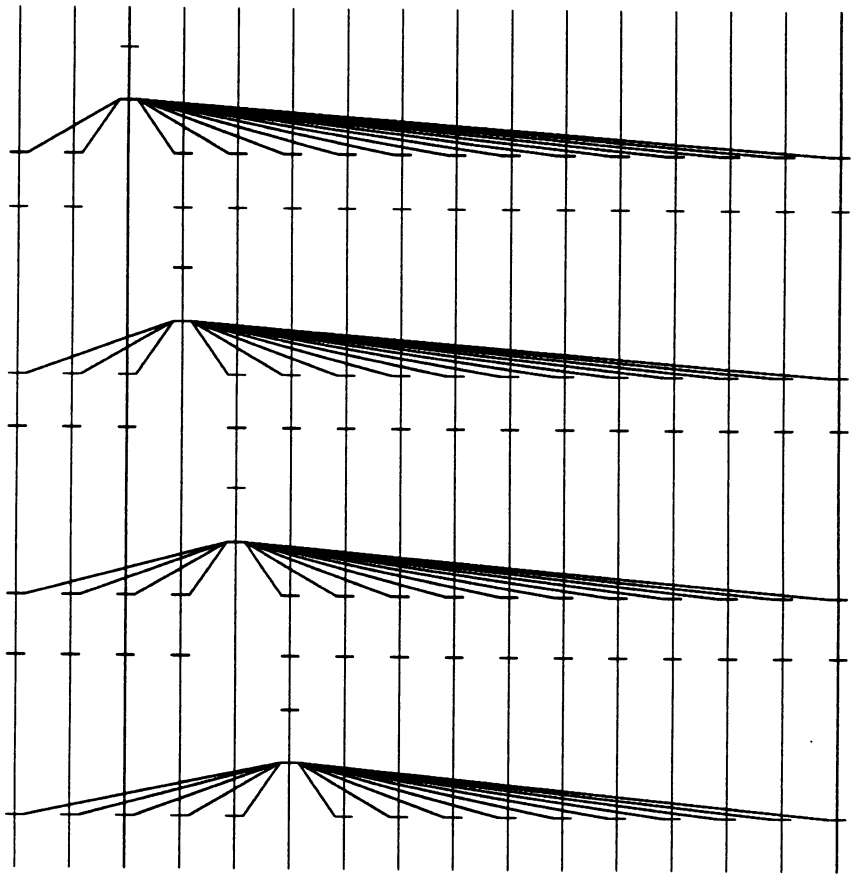

Figure 2: Rounds in the Upper Triangulation Shown in Logical Time.

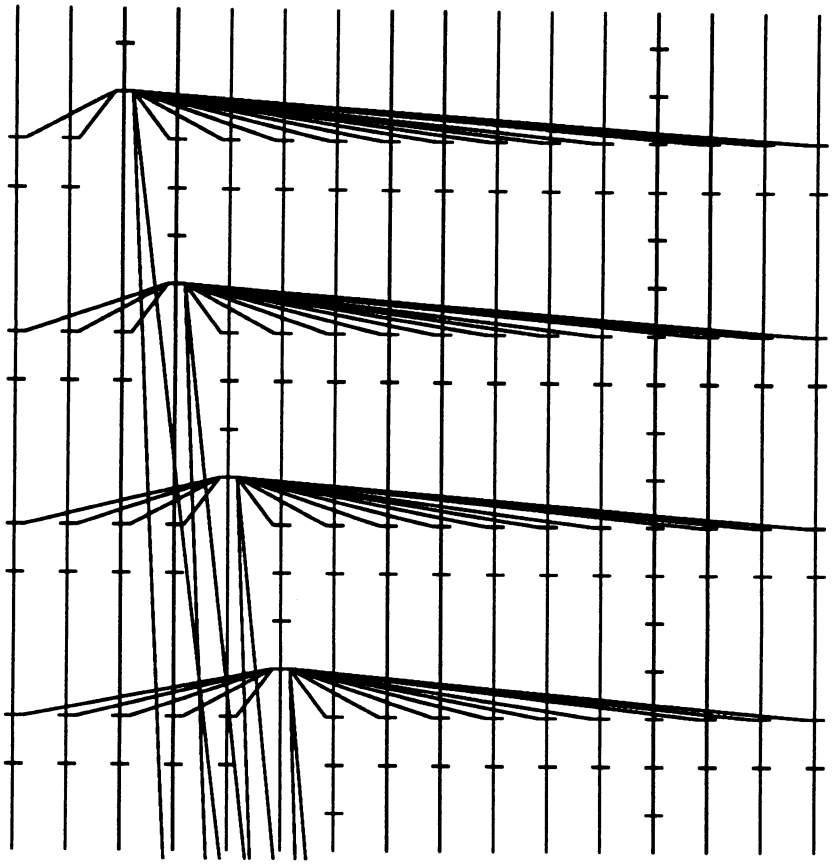

Figure 3: Rounds of an Erroneous Upper Triangulation Shown in Logical Time.

communication activity. Here we show how a variety of visualizations produced by Moviola are useful for understanding the communication relationships of processes.

Figure 2 illustrates several phases in the upper triangulation computation using a logical-time Moviola diagram. In each phase a process broadcasts a pivot row by copying it to a shared buffer. When other processes are ready to receive the pivot row, they each copy the data from the shared buffer into their own local buffer. In this visualization, the structure of broadcast communication becomes apparent; because the figure displays four rounds of the computation, the phase structure is also apparent.

Figure 3 shows the same slice of an execution produced by an incorrect version of the program, which contains an error in the polling protocol that checks to see if a buffer is full of data. The error allows a process to falsely conclude that a buffer is full when it is not. The figure shows that process 13 (the fourth process from the right) is not participating correctly in the receipt of broadcasts of the pivot row. This process has an event in each logical time step, but none of these events are connected to others by arcs. In each round we see two steeply sloped arcs indicating that each pivot row is received twice at a later time. Examination of a later slice of the space-time diagram (not shown) indicates that process 13 later receives two copies of each row. From the figure it is clear that the pattern

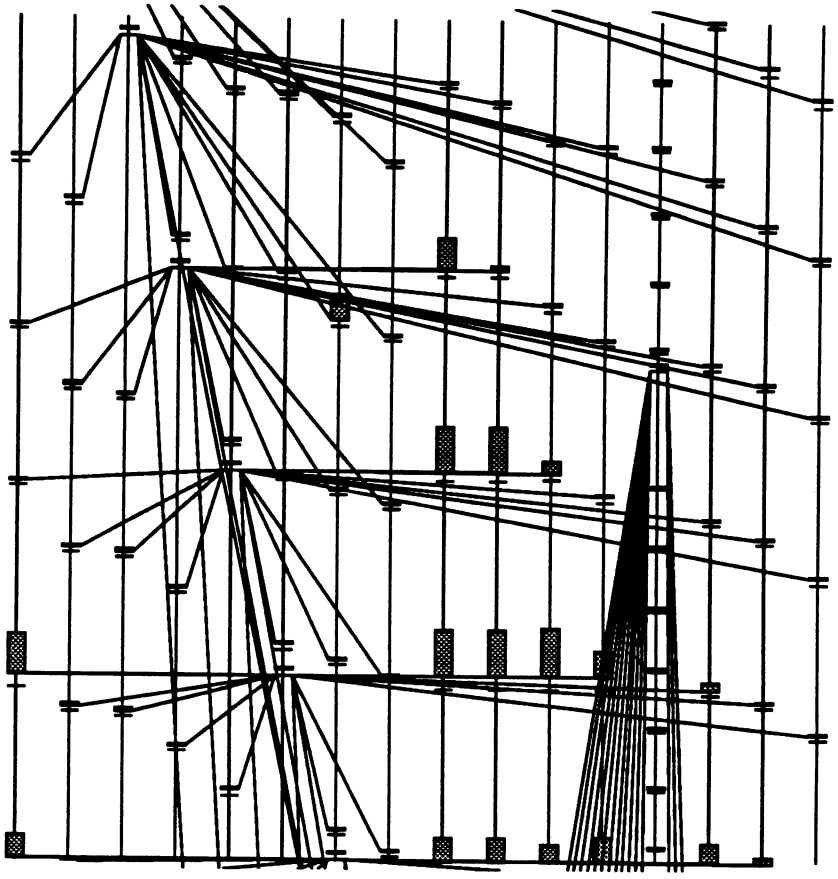

Figure 4: Rounds of an Erroneous Upper Triangulation Shown in Physical Time.

of communication exhibited by process 13 is erroneous, although the cause of the problem is not readily apparent.

Since logical time is insufficient to diagnose the problem, we move to a physical time view. Figure 4 shows the same pivot row broadcasts in physical time. In this view we can see that the broadcast phases that appear separate in logical time actually overlap in physical time. We also see that process 13 broadcasts during this interval. The algorithm specifies that processes should broadcast pivot rows in turn, so process 13 is clearly far ahead of the others.

Using Moviola, we focus on the details of the events that precede the broadcast by process 13. Moviola enables us to view the raw synchronization information associated with each event. Synchronization events are tagged with version numbers of the objects to which they refer. From these version numbers, we see that process 13 copies pivot row buffers before they are filled. This sequence of visualizations thus leads us to examine the communication buffer synchronization code to pinpoint the problem. We discover that an error in initialization causes the first process to access the buffer to believe the buffer is full when it is not. We also discover that process 13 is executing on a faster processor (our multiprocessor configuration includes several MC68000 processors and one MC68020

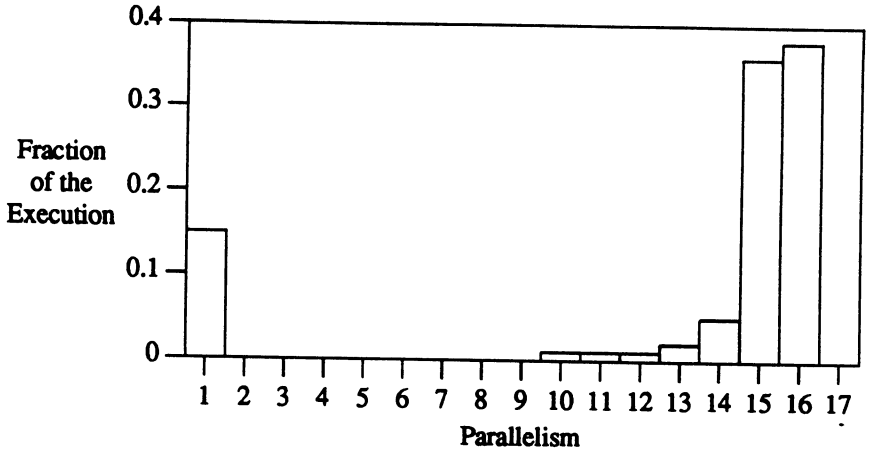

Figure 5: Distribution of Parallelism for the Upper Triangulation Program.

processor), and therefore is always the first process to access each buffer.

5.2 Performance Analysis

Figure 5 shows an anonymous-process view of an execution of the upper triangulation program on 17 processors. This histogram provides an abstract view of the performance of the program by displaying the fraction of the total execution time spent at each degree of parallelism. The information displayed in this view is computed directly from the execution history by projecting the number of active processes on each cut through the graph in physical time onto a single time scale. This figure shows that 15 or more processes are active most of the time; however, about 15% of the time, only one process is active.

To examine the distribution of parallelism in more detail, we examine how parallelism is distributed in time over the execution. Figure 6 plots the parallelism utilized at each point in the execution. This graph relates the degree of parallelism seen in the previous figure to physical time during execution. Using this figure, we can attribute most of the lack of parallelism to program startup. We also see that the degree of parallelism drops dramatically at fairly regular intervals. In most phases there is no appreciable loss of parallelism, but occasionally the program appears to encounter barrier synchronization.

One drawback of figure 6 is that every change in the degree of parallelism is presented; frequent changes near the end of the execution are obscured in the display. Figure 7 averages the degree of parallelism over 40 evenly spaced intervals. This figure provides a clearer indication of the average degree of parallelism as execution progresses. Although figure 6 shows that the degree of parallelism frequently dips as low as 1 near the end of the execution, figure 7 shows that the average degree of parallelism is still above 13.

The degree of parallelism at the beginning of the execution shown in figure 6 suggests a serial component at the start of the execution. Figure 8 shows a space-time diagram of the beginning of the execution. This view reveals that the master process (in the leftmost

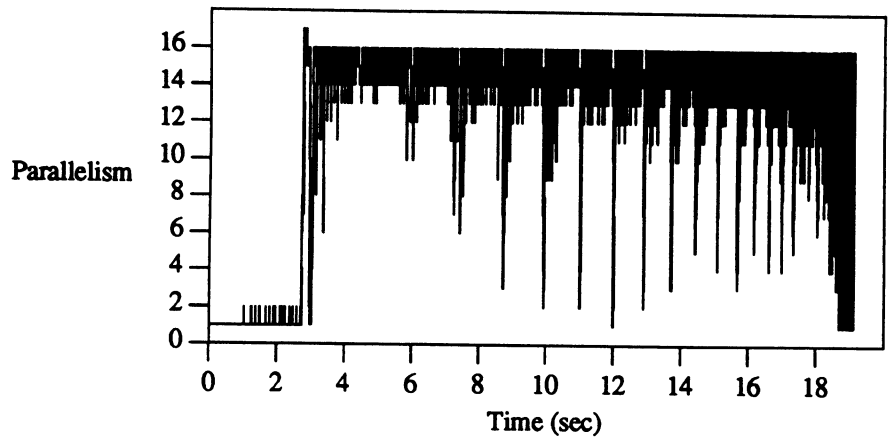

Figure 6: A Trace of Utilized Parallelism.

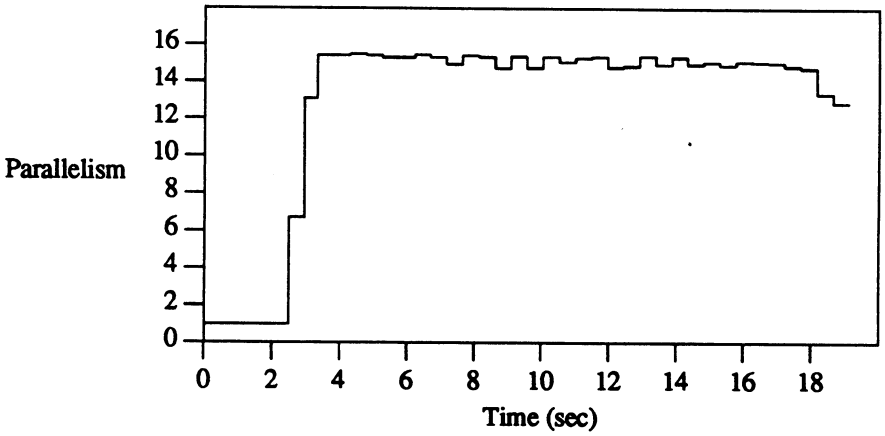

Figure 7: An Averaged Trace of Utilized Parallelism.

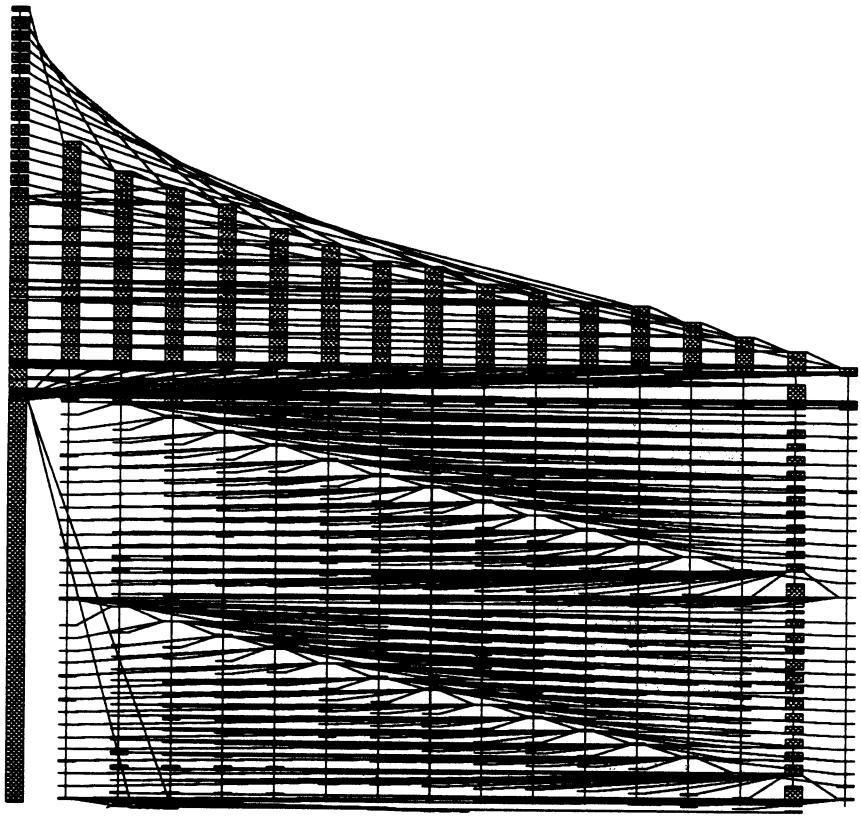

Figure 8: Startup and First Two Cycles of the Upper Triangulation.

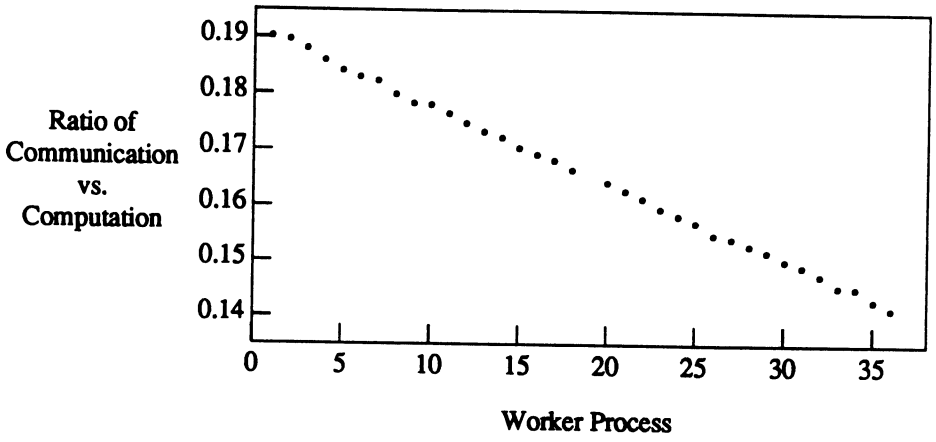

Figure 9: Ratio of Communication vs. Computation.

column) creates each worker process in sequence. Also evident in this figure is that one process, the second from the right, spends a significant fraction of its time waiting during each round of the computation. (Wait time is represented by a shaded box.) A visual inspection of the data partitioning code reveals nothing unusual—the static assignment of matrix rows to processes is balanced. There is no explanation for this anomaly, other than the relative speed of processes. We discover that, once again, one process is executing on a faster processor than the others. The fact that this process performs its elimination steps nearly twice as fast as other processes accounts for the large fraction of the execution time spent with 15 active processes. One process spends nearly half its time waiting, thus reducing by one the degree of parallelism for a large fraction of the execution.

One important measure of parallel program performance is speedup, a measure of how much the execution time decreases with an increase in the number of processors. Speedup is a function of how much time each processor spends actually working on the problem compared with time spent on process, communication, and synchronization overhead. A useful visualization for determining upper bounds on program speedup is a graph of the ratio of communication time to computation time for each process, averaged over the entire execution. We can define communication time in each process to be the sum of the lengths of the intervals during which the process holds a lock on any shared data structure. Similarly, computation time can be defined as the time during the execution during which no locks are held. Based on these definitions, figure 9 shows the ratio of communication to computation for each process in a 36 process execution of our program, omitting the data for the faster processor. This graph indicates that a substantial fraction of the execution time is spent waiting for pivot rows to become available and then copying them between processors.

The graph also shows a puzzling correlation between process number and communication ratio, which ranges from a high of 19% for the first process to a low of 14% for the last process. Because each processor is responsible for n/p rows, we expect the ratio of communication to computation to be identical for each of the processes. We also expect this ratio to remain fairly constant throughout the execution.

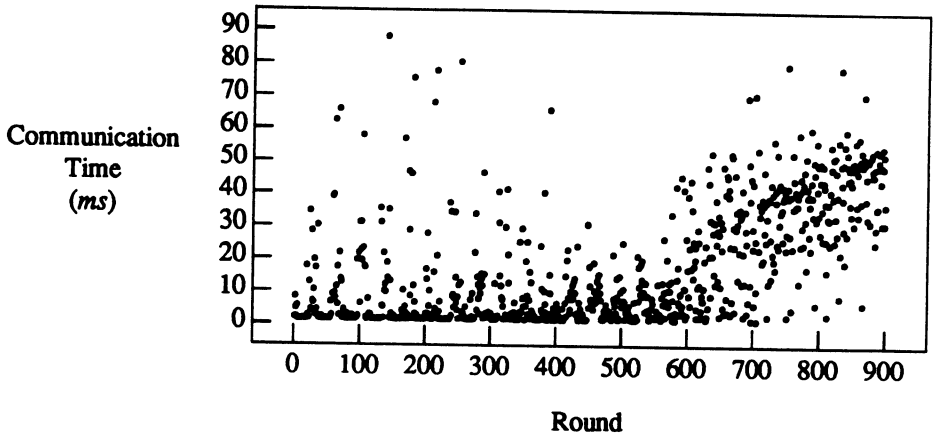

Figure 10: Communication Time per Round for Worker Process 1.

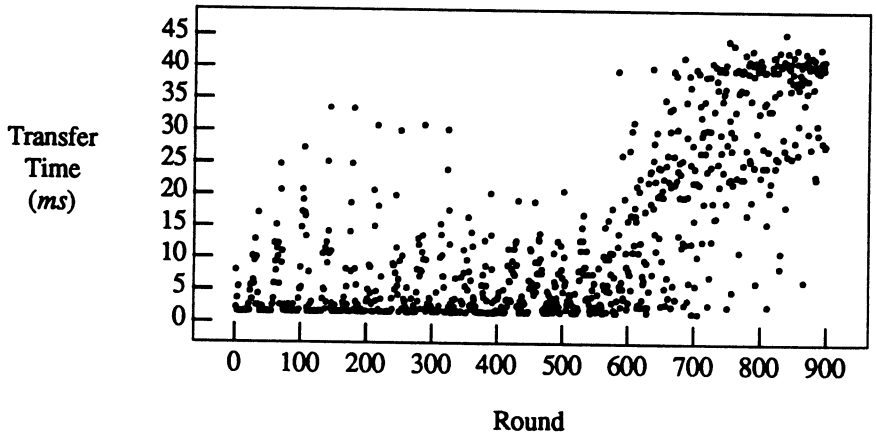

Figure 11: Pivot Row Data Transfer Time per Round for Worker Process 1.

To examine the relative imbalance in communication ratios, a more detailed view is created by displaying communication time as a function of phase time. To create this view, a tool traverses the execution history and records the amount of communication time per process per phase. Figure 10 shows a trace of the communication time per phase for the process with the highest ratio of communication to computation. The graph shows a trend towards increasing communication time in the later rounds, although there is also a large variability in the communication time within a round. To understand the cause of this trend, we separate the two sub-intervals associated with each communication operation: data transfer time and waiting time. Figure 11 shows the data transfer time for this process. Comparing with figure 10, we see that the data transfer time accounts for the largest part of the communication time in the later rounds. Examination of the program reveals that the amount of communication (the size of a pivot row to be transferred) does not decrease with the amount of computation; the program continues to transfer entire rows, even entries

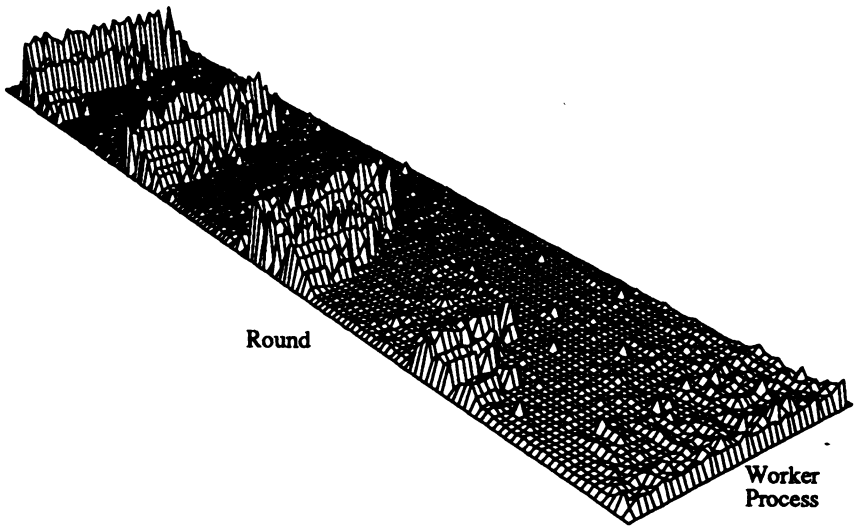

Figure 12: Pivot Row Data Transfer Time per Round for All Workers (150 Rounds).

known to contain zero. Since the amount of data transferred in each round is constant, the increase in transfer time is due to increased memory and switch contention. To minimize communication costs, we can avoid transferring entries that have been eliminated, thereby reducing contention and improving overall performance by 13%.

We still have not explained the nearly periodic drop in the degree of parallelism in Figure 6. An examination of the space-time diagram of the improved program shows that there is significant waiting in the early rounds, indicating imbalance in the computation. The static partitioning scheme, in which an equal number of rows of the matrix are assigned to each processes, suggests that the computation should be balanced. Figure 12 plots the data transfer time for each of the 36 worker processes for 150 rounds of the computation (upper triangulation of a 900×900 matrix). Rounds in the computation flow from foreground to the rear of the plot (along the Y axis). Pivot row transfer time in each round is plotted on the Z axis. Workers are distributed along the X axis with worker 1 at the left and worker 36 at the right. The display dramatizes how contention periodically builds during each cycle of the computation. A triangular wavefront of contention rises above the plane that corresponds to minimum transfer time. This visualization leads us to an explanation of the problem. After a process broadcasts a pivot row, it manipulates one fewer row for the remaining rounds in the cycle. This imbalance in work causes processes that broadcast early in each cycle to get ahead of processes that broadcast later in the cycle. When several processes are ahead of the process computing the pivot row, contention occurs as they simultaneously initiate a data transfer once the pivot row is available. Thus, as each cycle progresses, the number of processes that must wait for the pivot row increases, producing a barrier effect.

It is clear that the imbalance present in our improved program is a characteristic of any static partitioning of the problem. A dynamic partitioning strategy (see [4] which discusses a dynamic load balancing scheme for Gaussian elimination) offers a chance to eliminate some of this imbalance. Armed with this information, we can make intelligent choices about how

to improve performance as the program is applied to larger problem sizes or is executed on a larger number of processors.

6 Conclusions

Parallel program analysis is essentially a problem in information management. The amount of execution state that must be examined is overwhelming, so flexible techniques for selecting and presenting information are crucial. Any general solution must provide multiple views of an execution and allow the user to tailor the visual display to specific analyses.

All views are derived from a three-dimensional space of process interactions, process states, and time. Debugging and performance tuning require that we navigate through this space. We have described a methodology that structures the sequence of views to be examined during the program development cycle. The methodology is supported by a programmable toolkit that facilitates the creation of multiple views and user-defined visualizations. Our experience has shown that the ability to generate views in an integrated environment makes it easy to take advantage of many different views, which greatly simplifies analysis.

7 Acknowledgments

We thank Ivan Bella, the primary implementor of Moviola, and Neil Smithline for assisting in the construction of the toolkit. We also thank Robert Hood of Rice University for his help in constructing the contour visualization of the matrix in our case study. The contours were created using a tool he developed as part of the Parascope programming environment.

References

[1] R. A. Becker and J. M. Chambers. *S: An Interactive Environment for Data Analysis and Graphics.* Wadsworth Advanced Book Program, Belmont, CA, 1984.

[2] M. H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988. Ph.D. Thesis, Department of Computer Science, Brown University.

[3] A. L. Couch. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* PhD thesis, Department of Computer Science, Tufts University, Apr. 1988. Available as Tech. Rep. 88-4.

[4] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node Butterfly parallel processor. In *Proc. 1985 International Conference on Parallel Processing*, pages 531–540, St. Charles, Illinois, Aug. 1985.

[5] T. A. DeFanti, M. D. Brown, and B. H. McCormick. Visualization: Expanding scientific and engineering research opportunities. *IEEE Computer*, 22(8):12–26, Aug. 1989.

[6] R. J. Fowler and I. Bella. The programmer's guide to moviola: An interactive execution history browser. Technical Report 269, Department of Computer Science, University of Rochester, Feb. 1989.

[7] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, Madison, WI, May 1988. Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[8] A. A. Hough and J. E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proc. 1987 International Conference on Parallel Processing*, pages 735–738, Aug. 1987.

[9] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):124–150, May 1987.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] R. J. LeBlanc and A. D. Robbins. Event driven monitoring of distributed programs. In *Proc. 5th International Conference on Distributed Computing Systems*, pages 515–522, Denver, CO, May 1985.

[12] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[13] T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C. E. Fineman. Visualizing performance debugging. *IEEE Computer*, 22(10):38–51, Oct. 1989.

[14] S. P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, Mar. 1985.

[15] Z. Segall and L. Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, 2(6):22–37, Nov. 1985.

[16] D. Socha, M. L. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. In *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 206–215, Madison, WI, May 1988. Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[17] J. M. Stone. A graphical representation of concurrent processes. In *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 226–235, Madison, WI, May 1988. Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[18] L. D. Wittie. Debugging distributed C programs by real time replay. In *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 57–67, Madison, WI, May 1988. Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[19] C.-Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. *Proc. 8th International Conference on Distributed Computing Systems*, pages 366–375, June 1988.

[20] T. Yuasa and M. Hagiya. Kyoto common lisp report. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1985.